

Métodos de Pesquisa Heurística para Resolução de Problemas

Jogo: “Folding Blocks”

Trabalho realizado por:

Bernardo Moreira – up201604014

Filipe Nogueira – up201604129

Francisco Pereira – up201605306

Introdução

- Neste projeto é pretendido implementar diversos algoritmos de pesquisa capazes de vencer uma série de níveis do jogo “**Folding Blocks**”.
- Este jogo, inicialmente composto por um tabuleiro com uma ou mais peças, tem como objetivo, ocupar todos os espaços dos tabuleiros utilizando essas peças. Esta utilização é feita através da seleção da peça a jogar, seguida de uma transformação simétrica dessa mesma peça de acordo com o operador selecionado, isto é, cada movimento vai **duplicar** o tamanho da peça na direção escolhida.

Formulação do Problema

- **Representação do estado:** O tabuleiro do jogo é representado por uma matriz com tamanho variável (de acordo com o nível). Esta matriz é do tipo `int[][]`, cujo valores podem ser:
 - 0 - caso de espaços vazios;
 - N - sendo $0 < N < \text{Tamanho máximo do tabuleiro}$, onde a cada N corresponde um tipo de peça.
- **Teste objetivo:** O jogo acaba quando o tabuleiro não tiver espaços vazios. Ou seja, quando em todas as posições do tabuleiro x e y, `board[x][y]` seja diferente de 0.
- **Operadores:**

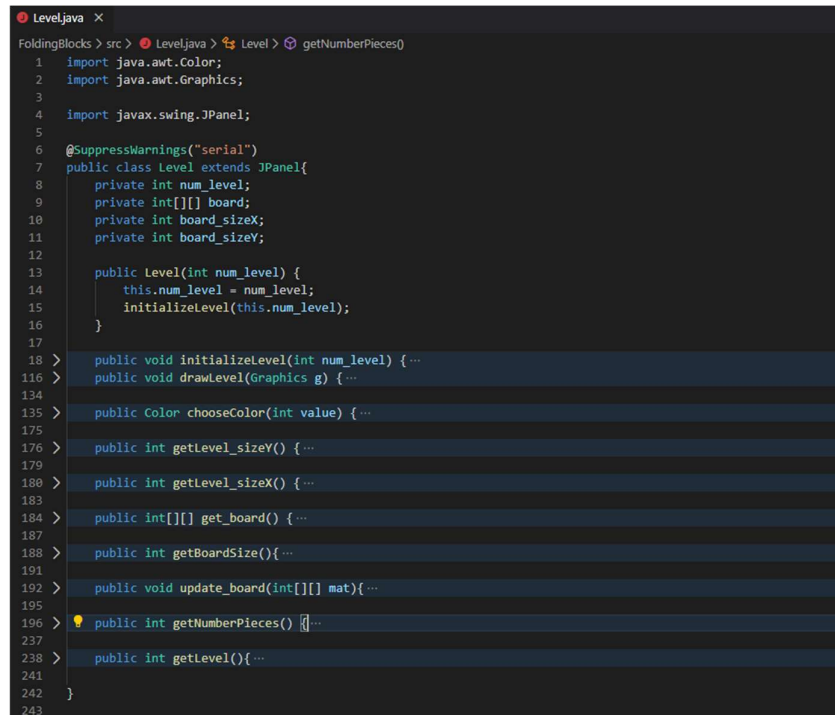
Nomes	Pré-condições	Efeitos	Custos
Dobrar para a Esquerda	$Yb \geq 0$; $Tab[xi][yb] = 0$;	$Dist = (yi - y_eixo) + 1$; $Yb = y_eixo - Dist$; $Tab[Xi][Yb] = ID$;	1
Dobrar para a Direita	$Yb < M$; $Tab[xi][yb] = 0$;	$Dist = (y_eixo - yi) + 1$; $Yb = y_eixo + Dist$; $Tab[Xi][Yb] = ID$;	1
Dobrar para Cima	$Xb \geq 0$; $Tab[xb][yi] = 0$;	$Dist = (xi - x_eixo) + 1$; $Xb = x_eixo - Dist$; $Tab[Xb][Yi] = ID$;	1
Dobrar para Baixo	$Xb < N$; $Tab[xb][yi] = 0$;	$Dist = (x_eixo - xi) + 1$; $Xb = x_eixo + Dist$; $Tab[Xb][Yi] = ID$;	1

- ID - referente à peça a ser jogada;
- `Tab[][]` - valor na posição de cada tabuleiro (pode ter os seguintes valores: 0 - se estiver livre / ID - numero referente à peça a ser jogada);

- X_i / Y_i - coordenada referente à linha/coluna (respetivamente) do bloco atual;
- X_b / Y_b - coordenada referente à linha/coluna (respetivamente) do bloco novo;
- x_eixo / y_eixo - coordenada referente à linha/coluna (respetivamente) do eixo de simetria;
- Dist - distância ao eixo de simetria;
- N x M - dimensões do tabuleiro de jogo (linhas/colunas);

Implementação do Jogo

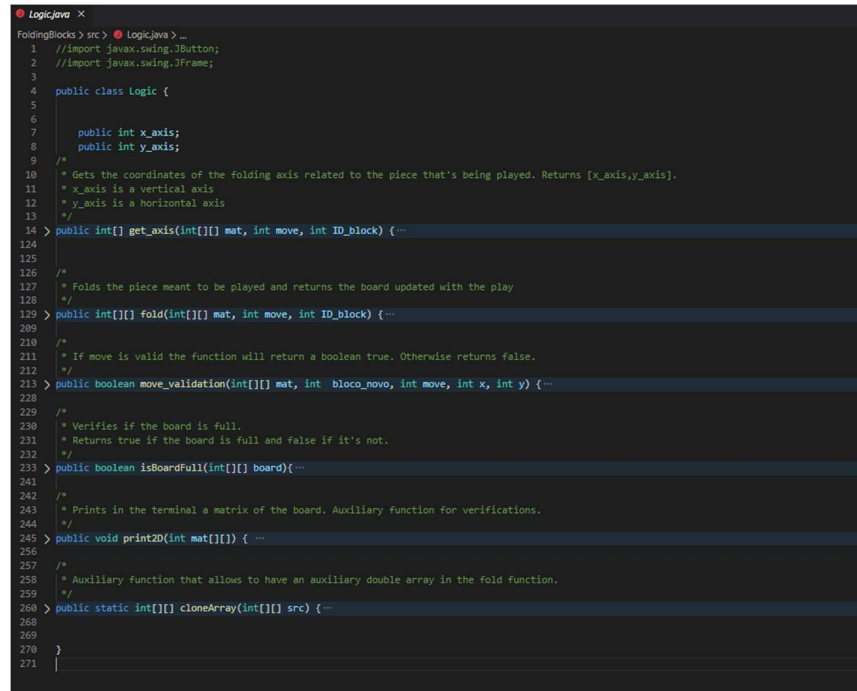
- Linguagem escolhida para desenvolvimento do código: **Java**
- Trabalho realizado em **Eclipse & VSCode**
- Código foi dividido pelas seguintes classes:
 - Main.java
 - Game.java
 - Level.java
 - Logic.java
 - Algoritmo.java
- A classe Level trata de definir, e desenhar os níveis. Trata da atribuição de cores consoante os valores da matriz e trata de atualizar a mesma.



```

Level.java
FoldingBlocks > src > Level.java > Level > getNumberPieces()
1  import java.awt.Color;
2  import java.awt.Graphics;
3
4  import javax.swing.JPanel;
5
6  @SuppressWarnings("serial")
7  public class Level extends JPanel{
8      private int num_level;
9      private int[][] board;
10     private int board_sizeX;
11     private int board_sizeY;
12
13     public Level(int num_level) {
14         this.num_level = num_level;
15         initializeLevel(this.num_level);
16     }
17
18     > public void initializeLevel(int num_level) { ...
116 > public void drawLevel(Graphics g) { ...
134
135 > public Color chooseColor(int value) { ...
175
176 > public int getLevel_sizeY() { ...
179
180 > public int getLevel_sizeX() { ...
183
184 > public int[][] get_board() { ...
187
188 > public int getBoardSize() { ...
191
192 > public void update_board(int[][] mat){ ...
195
196 > public int getNumberPieces() { ...
237
238 > public int getLevel(){ ...
241
242 }
243
  
```

- A classe Logic verifica a lógica de jogo. Trata de fazer as jogadas, bem como as verificações de jogada e de fim de jogo.



```

Logic.java
FoldingBlocks > src > Logic.java > ...
1 //import javax.swing.JButton;
2 //import javax.swing.JFrame;
3
4 public class Logic {
5
6
7     public int x_axis;
8     public int y_axis;
9
10    /*
11     * Gets the coordinates of the folding axis related to the piece that's being played. Returns [x_axis,y_axis].
12     * x_axis is a vertical axis
13     * y_axis is a horizontal axis
14     */
15    > public int[] get_axis(int[][] mat, int move, int ID_block) { ...
16
17
18    /*
19     * Folds the piece meant to be played and returns the board updated with the play
20     */
21    > public int[][] fold(int[][] mat, int move, int ID_block) { ...
22
23
24    /*
25     * If move is valid the function will return a boolean true. Otherwise returns false.
26     */
27    > public boolean move_validation(int[][] mat, int bloco_mov, int move, int x, int y) { ...
28
29
30    /*
31     * Verifies if the board is full.
32     * Returns true if the board is full and false if it's not.
33     */
34    > public boolean isBoardFull(int[][] board) { ...
35
36
37    /*
38     * Prints in the terminal a matrix of the board. Auxiliary function for verifications.
39     */
40    > public void print2D(int mat[][]){ ...
41
42
43    /*
44     * Auxiliary function that allows to have an auxiliary double array in the fold function.
45     */
46    > public static int[][] cloneArray(int[][] src) { ...
47
48
49    }
50
51    |

```

- Finalmente, tanto a classe Main como a Game servem para funcionalidades da interface. Permite a criação de uma janela para o jogo, bem como permite à classe Level que desenhe nesta mesma interface. Inicialmente implementamos também detecção de pressão de teclas, para testarmos as funcionalidades do jogo e permitir ao utilizador jogar o mesmo.

A classe Game dá também início ao algoritmo de pesquisa utilizado (conforme a escolha do utilizador) para a descoberta de solução, assim como interpreta as jogadas e as realiza autonomamente.

```

Main.java X
FoldingBlocks > src > Main.java > Main > main(String[])
1  import java.awt.AWTException;
2
3  import javax.swing.JFrame;
4
5  public class Main {
6
7      Run/Debug
8      public static void main(String[] args) throws AWTException {
9
10         if(args.length != 2){
11             System.out.println("Usage");
12             System.out.println("java Main <level><algorithm>");
13             System.out.println("<level> : [1-12] & <algorithmNumber> : [1-4]");
14         }
15         if(Integer.parseInt(args[0]) == 0 || Integer.parseInt(args[0])>12){
16             System.out.println("Invalid level");
17             return;
18         }
19         if(Integer.parseInt(args[1]) == 0 || Integer.parseInt(args[1])>6){
20             System.out.println("Invalid Algorithm");
21             return;
22         }
23         JFrame obj = new JFrame();
24         obj.setBounds(10, 10, 800, 700);
25         Game game = new Game(Integer.parseInt(args[0]), Integer.parseInt(args[1]));
26         obj.setTitle("Folding Blocks");
27         obj.setResizable(false);
28         obj.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29         obj.add(game);
30         obj.setVisible(true);
31     }
32 }

```

```

Game.java •
FoldingBlocks > src > Game.java > Game
22
23 public class Game extends JPanel implements KeyListener, ActionListener {
24
25     private final Timer timer;
26     private final int num_level;
27     private final Level l;
28     private Level level_algorithm;
29     public Algorithm al;
30     private int[][] mat;
31     private final int[][] board_ai;
32
33
34     private int move;
35     private int ID_block;
36     public boolean right;
37     public boolean left;
38     public boolean up;
39     public boolean down;
40     public Logic functional;
41     private final ArrayList<Integer> playsMove = new ArrayList<>();
42     private final ArrayList<Integer> playsID = new ArrayList<>();
43     private boolean ai;
44     private int ai_i;
45     private boolean end;
46
47
48     public Game(int levelNumber, int algorithmNumber) throws AWTException { ...
49
50     public void paint(final Graphics g) { ...
51
52     @Override
53     public void actionPerformed(final ActionEvent arg0) { ...
54
55     @Override
56     public void keyPressed(final KeyEvent e) { ...
57
58     @Override
59     public void keyReleased(final KeyEvent e) { ...
60
61     @Override
62     public void keyTyped(final KeyEvent arg0) { ...
63
64     public static int[][] cloneArray(final int[][] src) { ...
65
66     public void interpretPlays(final String plays){ ...
67
68     public void makeAIplay(int[][] board) throws AWTException{ ...
69 }

```

O objetivo de jogo, como já mencionado é completar todo o tabuleiro com as peças existentes no mesmo, e é possível através da nossa interface gráfica facilmente identificar o estado inicial e final do jogo:



Algoritmos e Heurísticas

Para a resolução do nosso projeto, de forma a testar a variação de tempo gasto na resolução do problema, número de nós explorados assim como número de jogadas feitas foram avaliadas 5 implementações:

- **“Depth First Search”**: É um algoritmo que tem como estratégia expandir sempre primeiro o nó de maior profundidade. Acaba por se apresentar como uma boa solução para o nosso problema em concreto dada a existência de várias soluções e a irrelevância da sequência em que as peças são jogadas. Aspectos negativos da implementação deste algoritmo no jogo “Folding Blocks” acontece quando este se depara com níveis mais complexos em que existe um maior número de jogadas que podem levar a resoluções impossíveis. Nestes casos o método de pesquisa prova-se ineficiente pois explora demasiados nós irrelevantes devido a falta de critério na seleção destes mesmos.
- **“Greedy”**: Este algoritmo tem como estratégia expandir o nó que está mais perto da solução. Utiliza uma função heurística que devolve um custo estimado do caminho mais curto do estado n para o objetivo (de acordo com a função heurística que lhe é atribuída). É de notar o facto de que esta estratégia não devolve necessariamente a solução ótima.

$$f(n) = \text{heurística}(n)$$

Com este algoritmo foram utilizadas duas heurísticas diferentes:

- Heurística2:

$$\text{heurística2}(n) = \text{número de espaços preenchidos}$$

Esta heurística avalia o número de espaços preenchidos e verifica a chegada à solução assim que o valor de $h(n)$ igualar o número de espaços existentes no tabuleiro. Utilizado com um método de pesquisa “Greedy” procura constantemente um movimento que proporcione o maior número de peças jogadas.

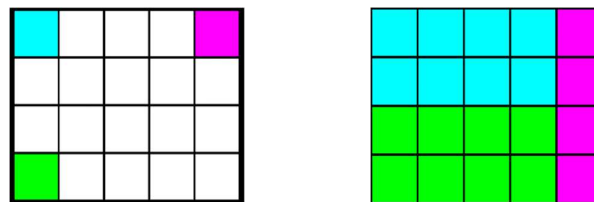
- Heurística3:

$$\text{heurística3} = \log_2 \left(\frac{\text{capacidade do tabuleiro}}{\text{espaços preenchidos}} \right) * \text{número de peças}$$

O valor retornado por esta função heurística estima, de acordo com o estado do tabuleiro, o número de jogadas até à solução final partindo do pressuposto que todas as peças terão o mesmo tamanho final. É aplicado ao algoritmo de pesquisa “Greedy” tentando sempre minimizar o número de jogadas até ao final da solução.

Esta função sobrestima a solução final, e como tal, não deve ser usada com o algoritmo A*.

Para o caso do nível 3, o estado inicial e final são os seguintes:



- **“A*”**: O algoritmo A* combina a pesquisa gulosa com a de custo uniforme minimizando a soma do caminho já efetuado com o mínimo previsto/estimado que falta até à solução. A solução do algoritmo A* é ótima e completa sendo importante o uso de uma heurística adequada.

$$f(n) = \text{heurística}(n) + \text{custo}(n)$$

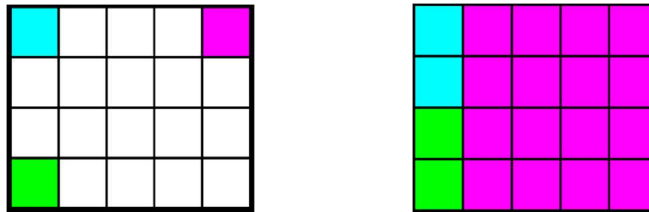
- Heurística4:

$$\text{heurística4}(n) = \log_2 \left(\frac{\text{capacidade do tabuleiro}}{\text{espaços preenchidos}} \right)$$

A função heurística em cima representada apresenta um valor subestimado do número de jogadas até encontrar solução final. Esta considera o melhor caso de jogadas para um tabuleiro com uma dada dimensão, que seria este ter apenas uma peça, e esta ser expandida ao máximo. Esta heurística apresenta-se admissível para ser usada com o algoritmo A* dado o facto de nunca sobrestimar a solução final, garantindo juntamente com o custo uma solução ótima ao problema.

É de notar que dadas as características do problema representa uma solução lenta e extensa, explorando por norma muitos nós.

Para o caso do nível 3, o estado inicial e final são os seguintes:



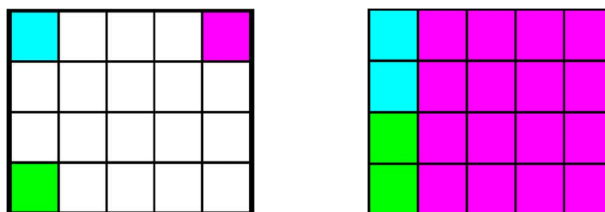
É possível notar que o algoritmo adota uma abordagem diferente daquela utilizada no algoritmo “Greedy”. Este opta por expandir ao máximo a peça cor de rosa, devido ao tamanho final maior que esta consegue obter, resultando numa solução com menos jogadas (solução ótima).

- **Implementação própria:** Neste caso o algoritmo atribui a cada nó o valor de acordo com a heurística em baixo representada e dá prioridade aquele com maior valor.
- Heurística5:

$$h5(n) = (\text{tamanho máximo da peça jogada}) + (\text{número de espaços preenchidos}) - (\text{custo})$$

Esta heurística tem por objetivo dar prioridade às peças que conseguem atingir uma maior peça total final (tendo na mesma em conta o novo número de peças que se obtém com cada jogada e o custo). Isto acontece, pois, a solução ótima obtém-se expandindo essa mesma peça ao máximo, dentro dos possíveis, e garantindo a existência de solução, pois vai ser possível cobrir mais espaços livres em menos jogadas.

Tal como o A*, este algoritmo encontra a solução ótima:



Comparação de Resultados

- Tempo:

TIME (ms)					
	DFS	Greedy H2	Greedy H3	A*	Our Own
Level 1	4	5	5	5	5
Level 2	4	5	5	9	5
Level 3	5	5	7	14	5
Level 4	5	5	6	7	5
Level 5	5	6	7	113	9
Level 6	5	7	29	95	7
Level 7	5	7	11	296	8
Level 8	7	10	9	971	8
Level 9	6	8	16	ND	715
Level 10	ND	ND	ND	ND	13284
Level 11	11	15	11366	ND	288

Através da comparação dos tempos podemos ver que a pesquisa em profundidade (DFS) e o algoritmo Greedy vão por norma apresentar maior rapidez na resolução dos problemas. Contrariamente o A* não se revela tão vantajoso nesse aspeto.

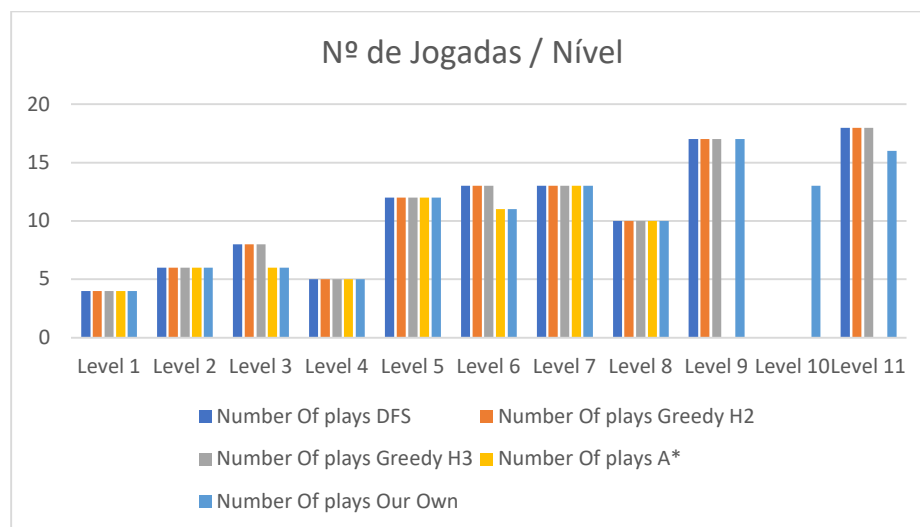
- Nós utilizados:

Number Of Nodes					
	DFS	Greedy H2	Greedy H3	A*	Our Own
Level 1	4	4	4	11	4
Level 2	6	6	9	60	6
Level 3	8	8	11	140	7
Level 4	5	5	7	34	5
Level 5	12	12	26	2374	26
Level 6	13	13	444	1931	13
Level 7	13	13	48	6651	27
Level 8	58	70	36	34871	25
Level 9	17	17	55	ND	22235
Level 10	ND	ND	ND	ND	191171
Level 11	97	85	217690	ND	5907

Por norma o valor de nós utilizados vai estar diretamente relacionado com tempo, dado que quantos mais nós o algoritmo explora mais tempo gastará a encontrar a solução. Podemos reparar que em geral a pesquisa para os algoritmos Greedy e DFS são menos extensas e como consequência mais rápidas, como já foi mencionado acima.

- Jogadas:

Number Of plays					
	DFS	Greedy H2	Greedy H3	A*	Our Own
Level 1	4	4	4	4	4
Level 2	6	6	6	6	6
Level 3	8	8	8	6	6
Level 4	5	5	5	5	5
Level 5	12	12	12	12	12
Level 6	13	13	13	11	11
Level 7	13	13	13	13	13
Level 8	10	10	10	10	10
Level 9	17	17	17	ND	17
Level 10	ND	ND	ND	ND	13
Level 11	18	18	18	ND	16



O número de jogadas utilizadas vai depender do nível em questão e do algoritmo utilizado. O A* vai oferecer sempre a melhor jogada perante o nível que lhe é apresentado, no entanto, acontece que em alguns níveis o número de jogadas até à solução é fixo, mesmo havendo várias soluções.

Conclusão

A escolha do algoritmo de pesquisa utilizado tem de ser ponderada e avaliada perante o parâmetro que pretendemos otimizar.

Se o interesse do utilizador for rapidez, então o algoritmo DFS ou Greedy será uma boa escolha e até certo nível escalável, sendo que para níveis onde os tabuleiros de jogo são bem maiores mantém na mesma um tempo de resposta rápido.

Se o interesse residir na capacidade de terminar o jogo no menor número de jogadas possíveis, então, nesse caso A* apresenta-se como uma boa opção, garantindo sempre o menor número de jogadas.

O algoritmo por nós implementado apresentou resultados interessantes, tendo sido o único a conseguir completar todos os níveis em tempos definidos e com rapidez, garantindo sempre a solução ótima.

Referências

- GeeksforGeeks:
 - <https://www.geeksforgeeks.org/greedy-algorithms/>
 - <https://www.geeksforgeeks.org/search-algorithms-in-ai/>
- Red Blob Games:
 - https://www.redblobgames.com/pathfinding/a-star/introduction.html?fbclid=IwAR0_pdUYGGfwkUHWjwAkKdxsq_A-1HqhF3XDlaMC0Wx95q_WESjoRgykIUU
 - <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>
 - <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#the-a-star-algorithm>
- Slides da Matéria lecionada em aula.

